A Short Isabelle/HOL Tutorial for the Functional Programmer

Thomas Genet

A Short Isabelle/HOL Tutorial for the Functional Programmer

Thomas Genet
genet@irisa.fr

Abstract

The objective of this (very) short tutorial is to help any functional programmer to quickly put its hand on Isabelle/HOL and catch a glimpse of its power. If you want some more afterward, please refer to the official Isabelle/HOL tutorial and the documentation available in the tool.

1 Installing and Starting Isabelle/HOL

Download Isabelle/HOL from `https://isabelle.in.tum.de/`. This tutorial is based on the 2020 version. If your operating system is neither old nor exotic, installation should be straightforward. Otherwise, you should refer to the installation instructions `https://isabelle.in.tum.de/installation.html`.

When Isabelle/HOL main window opens, it should be divided into three frames. The largest frame on the top left is used to enter programs and lemmas. Let us call it the **Theory** frame. The frame on the rightmost position gathers all built-in Isabelle's documentation. The frame at the bottom is for output of all program runs and proof goals. If the **Output** frame is not visible, click on the "Output" button on the bottom left of the window to make it visible.

2 Defining and Running Programs

In the Theory frame type the following:

```
theory Scratch
  imports Main
begin

end
```

where the name of the theory (here `Scratch`) should match with the file name (check that it is `Scratch.thy` in the leftmost topmost bar over the Theory frame). Then, between `begin` and `end`, we can define the elements of our theory: functions, tests, lemmas, etc. Let's start by our first program. Below `begin` and above `end`, type in the following code:

```
fun remove:: "'a => 'a list => 'a list"
where
"remove x [] = []" |
"remove x (y#ys) = (if x=y then ys else y#(remove x ys))"
```

Note that, unlike many programming languages, Isabelle/HOL does not use double quotes to denote strings but to delimitate function definitions, lemmas, etc. While typing this program you can notice two things. First, Isabelle/HOL replaces all combinations `=>` by the nicer symbol ⇒. Second, in the Output frame, Isabelle/HOL has automatically proven the termination of your program, by finding a termination order denoted by `"(λp. length (snd p)) <*mlex*> {}"`. Roughly, termination of `remove` is guaranteed because the length of the second parameter (i.e. the list) decreases for each recursive call. If no termination order is found, then you will see in the Output frame a red warning saying that automatic termination proof failed. The `fun` construct defines a function named `remove`, taking two parameters of type `'a` (any type) and `'a list` (list of elements of type `'a`) and whose result is of type `'a list`. The function is then defined using (possibly recursive) equations. Lists can be built using the empty list `[]` and the "cons" operator denoted by `#`. Lists can also be defined using shortcuts. For instance, the list `[1,2,3]` is a shortcut for `1#(2#(3#[]))`. Now let's try our function `remove`, which should remove all occurrences of an element from a list. Below the definition of `remove`, enter a simple test where `(* ... *)` is the syntax for comments:

```
value "remove 2 [1,2,3]"     (* this is a simple test *)
```

In the Output frame, you will see a red warning saying that there is a “Wellsortedness” problem. This is due to the fact that, by default in Isabelle, symbols “0”, “1”, etc. are not uniquely associated to the type `int`. Thus, evaluating `value "1"` will result in the same “Wellsortedness” problem. To denote the integer 1, we need to annotate the symbol “1” by the desired type: `value "(1::int)"` results in the integer 1. Let’s try again on our example:

```
value "remove (2::int) [1,2,3]"
```

Now, we have the expected result in the Output panel: `[1,3]::int list`. Note that specifying the type of one parameter was enough for the desired type to be inferred for the whole expression.

3 Automate the Search for Bugs

The next step is to check if our function contains errors. We could write more tests using `value` and verify, test after test, that the result fulfills our expectations. Instead, we define a property that our function is supposed to have, using a logical formula. For instance, type in the following sentence, where the ¬ logical symbol stands for “not”. To get the ¬ operator, type ~ character: you can then use the “Tab” key to select the symbol inside the selection box that opens[1].

```
lemma "¬ (List.member (remove e l) e)"
```

The function `List.member::'a list ⇒ 'a ⇒ bool` is part of Isabelle’s `List` library and results in the boolean `True` if the list contains the given element, and `False` otherwise. The formula states that the result of `remove e l` should **not** contain the element `e`. Since we provide no information on `e` or `l`, they are free variables. Thus, the formula we just defined is supposed to hold for all possible values of `e` (of type `'a`) and `l` (of type `'a list`).

The effect of the `lemma` command is to define the property and, at the same time, to switch Isabelle/HOL into proof mode. Below `lemma`, Isabelle/HOL waits for proof commands. In this mode you can no longer define functions nor state new lemmas. On the current lemma, we can first search for small finite counterexamples. The AutoQuickcheck may already have shown you a counterexample in the Output frame. Otherwise, this can be obtained by typing the `nitpick` command below the `lemma` line. While typing the first letters of the word `nitpick`, a selection appears in which one you can select the command you want to insert using the arrows of your keyboard and the “Tab” key. Once the `nitpick` command is inserted, it finds a counterexample to our lemma. With values `l = [a1,a1]` and `e = a1`, the lemma we stated is false. In other words, for any list `l` with two occurrences of the same element, the lemma is false. Let’s check `remove`’s behavior on a similar example. Typing in `value "remove (1::int) [1,1]"` permits to see that `remove` does not do the job: the result is `[1]` and our function is wrong. Click on the definition of `remove` and correct it. In the “then” part of the second equation, one should recursively call `remove` on `ys`, i.e. the second equation should be: `"remove x (y#ys) = (if x=y then (remove x ys) else y#(remove x ys))"`. Once corrected, you can click on the `value` line to check that the result is the expected one, and click on the `nitpick` line to check that no more counterexamples are to be found.

Now, let’s try to define a second property. You first have to close the ongoing proof. There are three ways to close the proof of a lemma: `done` when the proof is finished, `sorry` when you think that the lemma is correct but do not have completed the proof, and `oops` to close the proof mode when the lemma was shown to be false (if a counterexample is still found, for instance). In our case, we corrected the function, the lemma is likely to be true, but we do not yet have the proof. Thus, we type `sorry` below the `nitpick` command. With our last lemma we expect to show that `remove e l` removes all occurrences of `e` in `l` but what about the other elements in the list? Below `sorry`, enter the following lemma which checks that all the other elements have not disappeared from the list:

```
lemma "(length l) = (length (remove e l)) + (count l e)"
```

Note that you have to type a space character between `(length l)` and “=” to avoid Isabelle/HOL to replace the combination “)=” by the symbol “⊇”. In this lemma, the function `length::'a list ⇒ nat` returns the number of elements in a list (`nat` stands for natural numbers, i.e. non-negative integers) and `count::'a list ⇒ 'a ⇒ nat` counts the number of occurrences of a given element in a list. Thus, this lemma states that the number of elements in `l` is equal to the number of elements in `(remove e l)` plus the number of occurrences of `e` in `l`[2]. Again, the first thing to check is that there is no counterexample. Below the lemma, type `nitpick`. It finds a counterexample where `count` is considered as a free variable, as `l` and `e`. Furthermore, free variables are colored

[1] If you are looking for more logical symbols like ∧, ∨, ∀ or ∃, click on the “Symbols” button at the bottom of the main frame and then on the “Logic” or “Arrow” tabs. When you have found the needed symbol, you can either click to insert it or hover the mouse pointer over it to see the abbreviations you can type.

[2] We could state a stronger property, but let’s start by this one!

in blue, and so are `l`, `e` and `count`. Note that this is not the case for `length` and `remove`. The reason is that the function `count` is unknown. In fact, the function name in the library is `count_list`. This time, the counterexample is not due to a bug in the function but to an error in the lemma: it states a property on an unknown function `count` and `nitpick` invents a value for this function, falsifying the lemma. This permits to detect misspelling and such in the formulas. If you modify the lemma as follows, `nitpick` no longer finds counterexamples.

```
lemma "(length l) = (length (remove e l)) + (count_list l e)"
```

4 No more bugs? let's go for a proof

The fact that `nitpick` finds no counterexample does not imply that the property is true. To be sure that it holds we need to prove it using Isabelle/HOL proof commands. First, we try to prove this last lemma on `remove`, `length` and `count_list`. Since we know that there are no more simple counterexamples on the lemma, we can remove the `nitpick` command. Click on the lemma line and check the "Proof state" box on top of the output frame. The proof state consists, for the moment, of a unique proof subgoal which is exactly the lemma we stated. We are going to apply the main Isabelle proof command `auto` to see if this goal can be automatically simplified. Under the lemma, type `apply auto`. In the output frame, we can see that `auto` failed to simplify the proof goal. This is not surprising since `e` and `l` are free variables. In particular, we do not know if we can apply the first equation of the definition of `remove` since we do not know whether `l` is `[]` or not. To give more information to Isabelle about `l`, before the `apply auto` line, type `apply (induct l)`. Applying induction on `l` has split the proof goal in two subgoals:

```
1. length [] = length (remove e []) + count_list [] e
2. ⋀a l. length l = length (remove e l) + count_list l e ⟹
                    length (a # l) = length (remove e (a # l)) + count_list (a # l) e
```

The first subgoal corresponds to the case where `l` is known to be `[]`, i.e. `l` has been replaced by `[]` in the subgoal. In the second subgoal, `a` and `l` become new fresh variables, local to this formula. This is the meaning of the notation "⋀`a l.`" on the leftmost part of the formula. This second subgoal is built with Isabelle's deduction symbol "⟹". To solve a goal of the form $A \Longrightarrow B$, the objective is to prove B using the additional hypothesis A. This time, `apply auto` knows how to simplify those two goals. Click on the `apply auto` line below `apply (induct l)` to see its effect: there are no proof subgoals left. This concludes your first proof. You can now proudly close it using the `done` command. The complete lemma and proof should look like this in the Theory frame:

```
lemma "(length l) = (length (remove e l)) + (count_list l e)"
apply (induct l)
apply auto
done
```

Now, let's go back to the first lemma we wrote on `List.member` and `remove`, and prove it. You can remove the nitpick command. As above, an `apply auto` would not succeed because the definition of `remove` cannot be used to simplify the proof goal. Type `apply (induct l)` and then `apply auto`. This time, `apply auto` did not succeed to fully solve the subgoals. If you click on the `apply (induct l)` line and the `apply auto` line you can compare the subgoals to see the effect of `apply auto`. In the two subgoals, the expressions built on `remove` have been simplified but not the expressions built on `List.member`. This is due to the fact that the equations defining `member` are not visible for `auto`, because they are outside of the current theory. We could search in Isabelle/HOL theories for the name of the definition or lemma to apply. Instead of this, we can delegate this job to Sledgehammer. Sledgehammer can find and use functions definitions and lemmas outside of the current theory and call external theorem provers to solve proof goals. Click on the `apply auto` line, then click on the "Sledgehammer" button on the bottom left of the window to make it visible. In this Sledgehammer window, you find the list of automatic provers that are going to be used to solve your goal: `cvc4`, `z3`, etc. Click on the "Apply" button. After some seconds, you can see proofs proposed by the external theorem provers. In particular, you should see this result:

```
 Try this: apply (simp add: member_rec(2))
```

by clicking on the proposed proof command `apply (simp add: member_rec(2))`, it is added to your proof and solves the first subgoal. Proofs obtained from Sledgehammer use either `simp`, `metis`, `smt`, ... proof commands and lemma or definition names. Here, the proof command `simp` uses the second equation of the `member` definition to solve the goal. By clicking on the "Output" button, you can check that the first subgoal was solved. The last subgoal can be solved in the same way using Sledgehammer again. The last proof command introduced by Sledgehammer should be of the form `by (simp add: member_rec(1))`, where `by` is a proof command that also closes the proof mode like `done`. Let's finish this part by proving a last lemma (where the implication symbol ⟶ can be obtained by typing `-->`):

```
lemma "(length (remove e l)) = (length l)  ⟶ ¬ (List.member l e)"
```

This property means that if the length of `(remove e l)` and `l` are equal, then `e` should not appear in `l`. Nitpick does not find any counterexample. We can start the proof by `apply (induct l)` and `apply auto` as usual. We end up with four subgoals. Sledgehammer can solve the first one, but not the second one. So we are stuck with 3 subgoals. Let's carefully look at the first remaining subgoal:

```
1. ⋀l. length (remove e l) = Suc (length l) ⟹ List.member (e # l) e ⟹ False
```

This goal is of the form $A \implies B \implies \mathit{False}$. In other words, with assumptions A and B we have to prove *False*. The only solution for this subgoal to hold is if the assumptions, separately or together, are false. Here, the first assumption is clearly false: `length (remove e l)` cannot be equal to `(length l)+1` (`Suc i` stands for the successor of natural `i`). In fact, we already know, from the previous lemma, how `length (remove e l)` and `(length l)` relate. However, for Sledgehammer to use this lemma, we need to name it. Click back on the definition of the previous lemma and add a name to it with a colon. For instance:

```
lemma count_remove: "(length l) = (length (remove e l)) + (count_list l e)"
```

Come back to the proof and click on the line where Sledgehammer was failing. Now, Sledgehammer succeeds on this goal, and you can see that the found proof uses the `count_remove` lemma. Note that, for a lemma `A` to be used in the proof of lemma `B`, the definition of `A` should come before the definition of `B`. To prove the two remaining subgoals, and conclude the proof, you can use Sledgehammer again.

5 Datatypes

Let's define the algebraic datatype of polymorphic binary trees with two constructors `Leaf` and `Node`. Pay attention to the double quotes. Then, construct an object of this type and associate it to the constant name `myTree`. Finally, define two functions on trees:

```
datatype 'a tree= Leaf | Node 'a  "'a tree"  "'a tree"
definition "myTree= Node (1::int) (Node 2 Leaf Leaf) (Node 3 Leaf Leaf)"

fun numNodes:: "'a tree ⇒ nat"
where
"numNodes Leaf = 0" |
"numNodes (Node e t1 t2) = 1+(numNodes t1)+(numNodes t2)"

fun subTree:: "'a tree ⇒ 'a tree ⇒ bool"
where
"subTree Leaf  Leaf            = True" |
"subTree t     Leaf            = False" |
"subTree t     (Node e t1 t2)  = ((t=(Node e t1 t2)) ∨ (subTree t t1) ∨ (subTree t t2))"
```

The function `numNodes` counts the number of nodes in a binary tree and the function `subTree` checks if a tree is a sub-tree of another, where the symbol "∨" (denoting "or") is obtained by typing `\/`. Note that, in the definition, the order of equations matters. For instance, to evaluate a call to the function `subTree` definition, the second equation is tried only if the first one does not apply, etc. We can test those functions on the constant we just defined:

```
value "numNodes myTree"
value "subTree (Node 3 Leaf Leaf) myTree"
```

The following lemma states the following property: if a tree `t1` is a sub-tree of `t2`, then the number of nodes of `t1` is lesser or equal to the number of nodes of `t2`. Note that the symbol ≤ is obtain by typing `<=`.

```
lemma subNum: "(subTree t1 t2) ⟶ (numNodes t1) ≤ (numNodes t2)"
```

To prove this lemma, we need to perform a proof by induction on trees because functions `numNodes` and `subTree` are recursively defined on trees. We have the choice between an induction on `t1` or on `t2`. However, since `subTree` is recursively defined on the second parameter, induction on `t2` is more likely to succeed. The proof can be carried out using `apply (induct t2)`, then `apply auto` and Sledgehammer.

6 Code exportation

You can export the code of Isabelle/HOL functions into various programming languages (SML, Haskell, OCaml and Scala) using the `export_code` directive. For instance, below all the function definitions and outside of a proof, you can type: `export_code remove numNodes subTree in Scala` whose effect is to export all the necessary material for the above functions to run in Scala.

7 Acknowledgements

Many thanks to Tobias Nipkow, Jasmin Christian Blanchette and all the developers on Isabelle's mailing list. Many thanks also to Pauline Bolignano, Benoît Boyer, Gurvan Cabon, Thomas Jensen, David Mentré, Jørgen Villadsen and Yannick Zakowski for feedback on the tutorial.